de novo transcriptome assembly using Trinity

Matt Johnson

July 9, 2015

1 Introduction

Trinity is a suite of software modules that tackle the problem of de novo transcriptome assembly. It was designed to work with data from the RNA-Seq Illumina platorm, which generates between 10 and 40 million reads between 70 and 100 base pairs long. Trinity arranges these short sequences into contiguous sequences representing gene transcripts, while taking care to identify alternative splicing forms and paralogous genes.

Trinity operates using three independent software modules:

1. Inchworm, which divides the reads based on commonly found sequence patterns and assembles initial contigs.
2. Chrysalis, which groups related contigs (such as splice forms) together into connected *de Bruijn* graphs.
3. Butterfly, which collapses these graphs and aligns the reads to them, distinguishing between splice forms and paralogs.

This tutorial will introduce the Trinity software, how to run it, and how to understand its output files.

2 Obtaining Trinity

Trinity is freely available from `trinityrnaseq.github.io`

Due to the limitations on RAM and CPU speed, you probably don't want to run Trinity on your own computer. However, you can download and build Trinity and its modules by following the instructions on the website.

As you will find in this tutorial, running Trinity is not nearly as difficult as finding appropriate hardware on which to run Trinity. If you do not have access to a high performance computer (at least 60 GB of RAM and 10 CPUs), the Trinity website has several useful resources. The Pittsburgh Supercomputing Center, the Data Intensive Academic Grid, and Amazon Cloud computing are all potential options.

3 Input Data

The input data for Trinity is Illumina RNA-Seq data, preferably paired-end. Trinity is also equipped to use strand-specific data, but in this tutorial we will use its default settings. In a previous session, you used Trimmomatic to trim and quality check reads, and re-pair the read files. It is these two files that you will use to run the Trinity assembly.

Before executing any program, especially one with many options, it is useful to check the documentation to see the basic requirements. To see the options for Trinity, enter `./Trinity.pl` from within the Trinity directory. The list of options is very long and is divided among the different modules of Trinity.

3.1 Required specifications

seqType Typically, you will use FASTQ files.

JM The max memory to allocate to Jellyfish, a Java module that speeds up the generation of k-mers from the reads.

left and right Specifies the two read files; the read_1 file is typically "left"

3.2 Useful options

CPU Allows you to specify multithreading, which speeds up Butterfly considerably.

output The default is to create `trinity_out_dir` in the current directory, and place all output files within it..

full_cleanup Trinity produces many output files, this eliminates all except the assembly file.

Additional options allow you to stop Trinity after Inchworm or Chrysalis, and fine-tuning of Butterfly options that are beyond the scope of this tutorial.

4 Running Trinity

Using the trimmed read files, set up a command line for Trinity using the required specifications. For example:

```
/path/to/Trinity.pl --seqtype fq --JM 10G --left reads_1.fq --right reads_2.fq --CPU 4
```

While it is running, open a new terminal window and use the command `top` to monitor Trinity's progress.

5 Output files

Trinity generates a large number of files within its output directory. You can use many of these files to monitor your Trinity run as it progresses. At the end of the run, some of the files will be completely empty. If you prefer not to see all of these files after completion, you can specify `--full_cleanup` on the command line.

5.1 Trinity.fasta

This is the most important file in the output: a FASTA file containing the assembled contigs. The next two sessions of the workshop will delve deeper into the specifics of the file and how to judge how well Trinity did.

5.2 Jellyfish

One of the recent speed increases for Trinity has been the implementation of the Jellyfish module. This splits the reads into all of the observed kmers while noting their abundance. There will be two files generated: `jellyfish.1.finished` and `jellyfish.kmers.fa`.

The first file should be completely empty at the end of the run. The second file contains all of the observed 25-base pair kmers as a FASTA file, with the kmer abundance as the sequence ID.

5.3 Inchworm

Inchworm takes the kmer catalog and begins to assemble them into contigs. By default, inchworm uses kmers of length 25. This cannot be changed within the Trinity package,

but you can change it up to length 32 if you run Inchworm separately.

Beginning with the most frequently observed kmer, Inchworm looks for kmers shifted one basepair in the 3' direction. That is, if the current k-mer is `ACCTGT`, it will look for k-mers that begin with `CCTGT`. There would be a maximum of four of these: `CCTGTA`, `CCTGTC`, `CCTGTG`, and `CCTGTT`.

Inchworm extends the current contig by considering the most frequently observed of these four options. When Inchworm runs out of ways to extend in the 3' direction, it proceeds in the 5' direction. When a kmer is used to build a contig, it is removed from the contig pool.

The contigs are deposited in another FASTA file: `inchworm.K25.L25.DS.fa`. The K25 refers to the kmer length, L25 means that contigs had to be at least length 25 to remain, and DS indicates this was not a strand-specific run. The headers in the fasta file are used by Crysalis.

The developers note that Inchworm is the most memory-intensive process within Trinity, because it must hold all kmers in memory. If computational resources are slim, you can reduce the complexity of the dataset with the option `--min_kmer_cov 2`. This will ensure that only kmers that occur more than once are considered for contigs. The default is that all kmers are considered. Generally, this will eliminate super rare transcripts and sequencing errors, but will not usually affect assembly quality.

5.4 Bowtie

A recently implemented step in Trinity uses the alignment mapping tool Bowtie with paired-end reads. It uses the paired-end information to "scaffold" two contigs together if paired reads span them. The execution of Bowtie results in several output files, including .sam and .ebwt files typical of Bowtie output.

These files are *not* to be confused with the output of alignReads.pl, which aligns the reads to the completed assembly.

5.5 Chrysalis

Inchworm is very efficient at assembling contigs, by ensuring that no two contigs can share a k-mer. However, this excludes the possibility of splice variants and paralogs. For transcript isoforms, this may mean that Inchworm produces one contig with exons 1 and 3, and a separate contig with exon 2.

Chrysalis searches for kmers of length 24 (k - 1) that may be shared among contigs, and collapses these. If you are monitoring Trinity, this will show up as the *GraphFromFasta* process. Bundled contigs are located in a fasta file `bundled.fasta` within the `chrysalis` subdirectory in the trinity output directory.

Collapsed bundles are located within the `GraphFromIwormFasta.out` file, and provide the seed for the construction of de Bruijn graphs in the next step of Chrysalis. Within the `chrysalis` directory, each de Bruijn graph is placed in its own directory. Individual reads are mapped to each bundle, resulting in a large assortment of graph-specific fasta files within each of these directories.

Upon completion, Chrysalis creates two files containing commands for Butterfly (one per component), within the `chrysalis` directory.

5.6 Butterfly

The main purpose of Butterfly is to reconstruct transcripts including alternatively spliced isoforms. It executes the commands in the `butterfly_commands.adj` file in order. This is good to know, in case there is some error, you can check the `butterfly_commands.adj.completed` file to find the most recently completed command, and begin again from there.

6 The Trinity Assembly

The most important file produced by Trinity is `Trinity.fasta` (It may also have a prefix if you used the `--outdir` option). This contains all of the assembled transcripts in FASTA format. It is a text file, but it is probably very large. The first item to discuss is how Trinity labels the transcripts.

Terminal 1: Example of Trinity.fasta

```
[mjohnson@treubia trinity]$ head Trinity.fasta
>c99_g1_i1 len=339 path=[45:0-296 341:297-312 44:313-338]
GGGAAATTCTCGAGTCAGAACTCACTGATTTGATGTCAGTTTATACTCACTACTAGTGTA
CAAAATAATTGAATCAACACACAGCATGACAGCAACTTAACATTCCTAATAGCAGCAGAG
GCTCAGGAATAAGTGTGACCAATCAGATTCCACTTCCTTATCAGTTATCACCAGTTAGAA
CCTCGTAATCCACAAGCACTGCAAGGCTCAGGACTGAATAACTTAAGTTCCATAAATGGT
GAACCCTGGCTCCAGCTATCATTGCTACGATTGATGATGTGTCCAAGATAATGAAGTATC
TGTCAACACCAGCATCAATCCTCTTTCATGACTTTGGCC
>c99_g1_i2 len=323 path=[45:0-296 44:297-322]
GGGAAATTCTCGAGTCAGAACTCACTGATTTGATGTCAGTTTATACTCACTACTAGTGTA
CAAAATAATTGAATCAACACACAGCATGACAGCAACTTAACATTCCTAATAGCAGCAGAG
GCTCAGGAATAAGTGTGACCAATCAGATTCCACTTCCTTATCAGTTATCACCAGTTAGAA
CCTCGTAATCCACAAGCACTGCAAGGCTCAGGACTGAATAACTTAAGTTCCATAAATGGT
GAACCCTGGCTCCAGCTATCATTGCTACGATTGATGATGTGTCCAAGATAATGAAGTATC
```

```
AATCCTCTTTCATGACTTTGGCC
>c100_g1_i1 len=261 path=[1:0-260]
TTGGCCGTGCGAACCGTTGGTGGAATGCAGCTCTGTAGTGGAATGTTTTTCGGTGGTGAC
GGAGACCGTGACTTCGGGTTGTGTTTCGACCTTCTGCTCGGCGCTTTGGGCGACGAAGAC
GATCGGCGTGAAGGGCGCTCGTCATGCCATATGTGGAGTTGAGCAGGGGAGTCTGAAGGT
ATTGGCAGGCTATTTGGGTTCAGGTAGCGACGGAGCTCTGGCGAAGGCACTTCTGTAGAA
GGAGGTGGCCGCTTGTCGTCG
```

Each sequence begins with a header, and in FASTA format is always indicated by the > sign. The header contains three values separated by spaces:

- Trinity transcript: `c99_g1_i1` (This may be slightly different in various Trinity versions)
- length of the transcript in bases
- The "Graph Path" which indicates how the *deBruijn* graph was resolved.

For the most part you can ignore the second two parts. Most bioinformatic software will only consider up until the first space when identifying transcripts. For Trinity transcripts, this is the transcript id, which itself is made up of three values separated by underscores.

The first two values represent the **component ID**, and the third value is the **isoform id**. An isoform is a version of a transcript from one particular component, and each component can have one or many isoforms. In the example above, component `c99_g1` has two isoforms: `c99_gi_i1` and `c99_g1_i2` while the other component `c100_g1` has just one component.

In biological terms, the component is analogous to the *gene* and the isoform is analogous to the *splice form*. For example, one transcript may have Exon 1, Exon 2, and Exon 3; another may have just Exon 1 and Exon 3. These transcripts originate from the same part of the genome, but are spliced together differently after transcription. This is not a perfect correspondence, however. Brian Haas has warned that other factors, including incomplete assembly, may generate many isoforms for one component. Alternatively, isoforms from different genes (paralogs) may be accidentally grouped into the same component.

Because of the similarity of isoforms with in a component, this may affect some of your analyses. Some of the summary statistics presented below are calculated at the component (gene) level and the isoform level.

6.1 Contig number

Several factors can influence the total number of contigs. When coverage is low, more contigs are generally better, indicating that the contigs represent real transcripts. At

higher levels of coverage, you want the number of contigs to decrease and approach the approximate number of genes in your organism.

Since the Trinity assembly is a basic FASTA file, you can simply count the number of header lines with the UNIX tool `grep`:

```
grep ">" Trinity.fasta | wc -l
```

6.2 Assembly Statistics

It is impossible to look at every contig produced by Trinity, as there may be many thousands. Instead, summary statistics are useful to determine the quality of the assembly. In this section we will consider length-based assembly statistics.

Trinity provides a simple script for calculating statistics on the assembled transcriptomes:

Terminal 2: TrinityStats.pl output

```
[mjohnson@treubia trinity]$ /opt/apps/Trinity/util/TrinityStats.pl NW-1.Trinity.fasta


################################
## Counts of transcripts, etc.
################################
Total trinity 'genes':  107269
Total trinity transcripts:      129355
Percent GC: 45.58

########################################
Stats based on ALL transcript contigs:
########################################

        Contig N10: 4341
        Contig N20: 3404
        Contig N30: 2769
        Contig N40: 2256
        Contig N50: 1772

        Median contig length: 355
        Average contig: 799.62
        Total assembled bases: 103435347


#####################################################
## Stats based on ONLY LONGEST ISOFORM per 'GENE':
#####################################################

        Contig N10: 3963
        Contig N20: 2956
```

```
        Contig N30: 2254
        Contig N40: 1652
        Contig N50: 1011

        Median contig length: 316
        Average contig: 610.09
        Total assembled bases: 65443259
```

At the top, there are basic length statistics. Notice that the statistics are broken down into *All Transcripts* and *longest isoform*.

Developed as a standard to measure genome assembly success, N50 is analogous to the mean contig length. However, it gives further weight to longer sequences.

Imagine that all contigs are sorted by length, and then you started adding all of the lengths beginning with the longest. When the running tally represented 50% of the total length of the assembly, the length of that contig is the N50.

If you want to get lengths greater than N50, such as N90, you can use the script n50.py on Treubia:

```
n50.py Trinity.fasta 70 80 90
```

You can put any integer after Trinity.fasta on the command line. This will calculate N70, N80, and N90 values at the isoform level.

6.3 Limitations

Because the length statistics were designed for genome assemblies, they may not be applicable to the assessment of transcriptome assembly. For example, in genome assembly, the ideal N50 would approach the size of an entire chromosome as you added more data. By contrast, increasing sequencing depth for a transcriptome would eventually produce diminishing returns on N50 length as contigs approach true transcript size.

What you really want to assess is the ability of the assembly to produce true transcripts. In the next session, we will address how to use transcriptome annotation.

7 Mapping reads to the assembly

The trinity assembly was made by connecting many reads together in a *de Bruijn* graph.

- What is the support for the assembly?

- For trinity components with multiple isoforms, which one should be chosen for further analysis?
- Which genes are most highly expressed?

To answer these questions, we need to map the original reads back to the assembly.

For this section, it is best to refer to the tutorial on the trinity website:

`http://trinityrnaseq.github.io/analysis/abundance_estimation.html`

In the tutorial, `$TRINITYHOME` refers to `/opt/apps/Trinity` on Treubia.

For the input data, you will want to use the Trinity.fasta file and the paired trimmed FASTQ files from Trimmomatic.

Here is a sample command line for the `align_and_estimate_abundance.pl` script:

Terminal 3: Align and estimate abundance

```
/opt/apps/Trinity/util/align_and_estimate_abundance.pl \
  --seqType fq --left paired_trimmed_1.fq --right paired_trimmed_2.fq \
  --est_method RSEM --aln_method bowtie --thread_count 12 --trinity_mode \
  --prep_reference --transcripts Trinity.fasta
```

Note that this example uses 12 CPU cores, so adjust accordingly.

The script will use the alignment program `bowtie` to map the reads to the transcriptome. When multiple reads overlap on the same transcript, this is referred to as *depth of coverage*. The program `rsem` summarizes the depth of coverage into statistics such as the Fragments Per Thousand base pairs Per Million Reads, or **FPKM**.

Exercise: The `RSEM.isoforms.results` file lists several statistics for each isoform, including length and FPKM. Can you find any components where the longest isoform is different from the isoform with the highest FPKM value?

The abundance estimation tutorial on the website also shows you how to use other scripts in Trinity to summarize the FPKM statistics in the whole assembly. Follow the examples to generate the FPKM threshold file.

8 Visualizing the Assembly

The `Trinity.fasta` file is simply a text file containing contig DNA sequences. To assess visually how the raw reads correspond to the contigs, you can visualize them using a genome viewer such as the Integrative Genome Viewer.

Download IGV from here: `http://www.broadinstitute.org/igv/projects/downloads/IGV_2.3.19.zip`

Windows users can execute the `igv.bat` file. MacOSX users should navigate to the IGV directory and execute the `igv.sh` command. You may need to first run: `chmod ax` igv.sh+.

On Treubia, you can just enter the command `IGV` from the terminal.

Once it has launched, you can then open your `Trinity.fasta` file by selecting *Open Genome from File...* from the *File* menu. You can navigate to the different contigs and zoom in to see the sequence. Used alone, IGV is not particularly useful until you align the raw reads to your assembly.

8.1 SAM and BAM files

Alignment and mapping tools such as bowtie generate files with extensions such as `.sam` or `.bam`. A the Sequence Alignment Map (SAM) format is standardized so that many programs can easily use it.

A full specification of the SAM format can be found here:
`http://samtools.sourceforge.net/SAMv1.pdf`

The Binary Alignment Map (BAM) format condenses information in a SAM alignment into a binary format. Rather than plain text, the file is written in bytecode. Not only does this make the file smaller, utilities that use BAM files can generally run much faster.

A lot of information is contained in the header regions of a SAM file. For our purposes, the most important specification is how the file is sorted. IGV will only accept *coordinate sorted* files.

The `align_and_estimate_abundance.pl` script unfortunately does not generate the coordinate-sorted BAM file anymore. Instead, you should use the program `samtools` to generate this file:

```
samtools sort -@ 12 -O bam -o bowtie.csorted.bam bowtie.bam -T sorting
```

This uses the program "samtools" to sort the bowtie.bam file that the Trinity script produced. The relevant options are `-@ 12` (to use 12 CPUs), `-O bam` (for writing the output as a BAM format) `-o bowtie.csorted.bam` (the name of the sorted file) and `-T sorting` (a prefix for temporary files that should be deleted when samtools is finished).

The command will probably take some time to finish. Once it's done, you can visualize the coverage in your transcriptome by loading the file into IGV.